SECURE DATA ACCESSING SYSTEM AND METHOD

Field of the Invention

The invention relates generally to a system for accessing a computer and its database and in particular to a system and method for accessing data, such as in a database, that is protected by/ located behind a typical computer security system, such as a typical firewall.

Appendices

The present application includes several Appendices that are incorporated into the disclosure and are part of the disclosure. Appendix A is 47 pages long and contains the source code of the various Java classes located on the client computer that are used to implement one embodiment of the invention. Appendix B is 8 pages long and contains the source code of the various Java classes shared by the client computer and application server computer that are used to implement one embodiment of the invention. Appendix C is 19 pages long and contains the source code of the various Java classes located on the application server that are used to implement one embodiment of the invention.

Background of the Invention

Modernly, computer systems are being put into place wherein the software and data on which the computer operates may be located at one or more disparate computer resource locations. In addition, the databases being accessed by the software may be located on a different computing resource and may be, for example, located on a corporate intranet so that the resources to implement and execute the software are distributed. To accomplish this distributed software execution, it is necessary to be able to track the locations of the various resources on the Internet, such as by the IP addresses, so that the software resources and the data from one or more databases can be retrieved as needed. For example, pieces of the software code may be

located at one location while the data associated with the software code may be distributed across the Internet.

The problem with the distributed computing set forth above is that the computer network of most corporate users is behind a security device or system, such as a firewall, and/or subject to proxy servers and other security measures that prevent users from easily accessing databases outside of the firewall. The purpose of the firewall is to protect the computer network from hackers outside of the system trying to get into the computer network and to avoid a user behind the firewall from downloading a potentially dangerous piece of code from the Internet. In addition, if the software is written in Java, the applet downloaded to the client can only interact with the server from which it was downloaded which is a fundamental security feature of Java. Thus, trying to connect through the Internet to distant servers cannot be done using Java applets with most protocols. The exception is the common hypertext transfer protocol (HTTP). By way of background, the HTTP protocol is the protocol used for all Internet traffic and it has over 10,000 ports. By specification, each port is traditionally used for certain types of traffic. For example, Oracle® database connections traditionally use port 1521. The general unprotected port for web-site Internet traffic, by default, is port 80. In addition, secure HTTP protocol traffic (HTTPS) is typically communicated over port 433.

In approaching this problem, there are well known typical approaches such as those described in the Sun Microsystems, Inc. Java 2.0 Enterprise Edition ("J2EE") specification. In particular, the application server would make the call to the database server and then the application server would obtain the results and create business objects on the application server itself. The objects would then be sent to the requesting client. In this mode of operation, the objects would be resident on the application server and would then be available for other clients to use if they so chose to do so. This has the benefit of not duplicating effort, because the application server would maintain the objects for all clients to use, and the client would transport the objects back as needed. While this may be efficient, it has certain disadvantages for a dynamic distributing computing environment, such as: (1) it requires that the applet downloaded to the client contain the classes necessary to create the intended objects; and (2) the server must

install the same classes on itself for creating the business objects intended for the client. These disadvantages make the system much less dynamic, and create other problems. For example, different clients may create different objects from the same resultset that is returned from the database and the classes required to create the objects for all of the clients may not be available to the application server.

In more detail, the clients that need a database connection may not be able to connect to the server from behind secure corporate intranets. Since corporate intranets, in general, are secured with firewalls, proxies and other security measures, the clients will have to tunnel through them to gain access to the external resources. Furthermore, most firewalls do not allow direct Internet Protocol (IP) traffic between the Internet and the internal network they are protecting. Thus, most organizations behind firewalls have a proxy server running that allows people inside the organization to access the Web. For a distributed computing environment across the Internet, it is necessary to allow users to access multiple databases through these security measures. Thus, it is necessary to provide a system and method for accessing data from behind a corporate security system so that the distributed computing system can be efficiently operated and for permitting a user behind a firewall to execute a distributed computer application, and it is to this end that the present invention is directed.

Summary of the Invention

The secure data accessing system and method in accordance with the invention overcomes the limitations associated with the typical systems. The secure data accessing system may be particularly useful for a distributed application/computing environment in which there is a need for extensive access to a variety of database servers. To accomplish the communication by tunneling through a security mechanism, such as a firewall, the well known HTTP protocol is used in combination with JDBC drivers. The JDBC drivers use a set of unique parameters for connecting to the database server. In addition, database servers, like all other servers, require the clients to connect to them through a particular port using client libraries and/or JDBC drivers. In particular, all JDBC drivers are required to conform to the JDBC API specifications developed

by Sun Microsystems, Inc. Since all JDBC drivers are written to Java specifications, all of them must implement the JDBC interfaces.

In a distributed application environment that uses Java, the applet or the stand alone application needs to communicate with the database server. Java applets are only allowed to communicate with the server that the applet is originated/downloaded from due to security restrictions imposed by the Java language specification. In a distributed application environment, it is more likely that the database server might be on a different server than the application server and these servers might not be reachable by the client using protocols other than HTTP. In addition, JDBC drivers provided by the database vendors connect to the database using sockets. Furthermore, applets are restricted from making direct socket connections to other servers for the same security concerns. Corporate intranets need to provide access to the Internet to their users and, by default or de facto standard, the HTTP protocol has become the preferred protocol. Therefore, the best way for the applets to communicate with the server in accordance with the invention would be to use the HTTP protocol in combination with JDBC drivers in order to be as compatible as possible with the proxy servers and network firewalls.

The HTTP JDBC driver system in accordance with the invention may also be used to solve a problem faced by systems that need to communicate with a database through a firewall, such as web services. The system may also be used effectively with other systems including peer-to-peer systems, in jini services, mobile devices with small footprints and any other usage that requires a thin client to obtain data from a data storage device through a firewall, a network address translator (NAT) or other security mechanism, in a dynamic powerful way. For example, a dynamic application system that may need to be connected to multiple databases that may be located behind security mechanisms such as the dynamic application system described at http://www.verticalsuite.com.

Unlike a more traditional approach using the J2EE specification (J2EE stands for Java 2.0 Enterprise Edition), the HTTP JDBC driver in accordance with the invention has significant advantages. Using the HTTP JDBC driver, the application server may create a result and sends

that to the client who then creates the objects as needed at the client. The advantages to this approach become apparent in a dynamic, web-services type of environment in that it allows clients to be very thin, and to unilaterally determine the way in which they receive data. The client then has the power to dynamically determine how it wants the results presented or treated. For example, the client could choose to create its own business objects, or it could simply manipulate the data and/or present it to the user.

The HTTP JDBC driver system in accordance with the invention may affect/be beneficial to many different existing systems. For example, the driver system in accordance with the invention may be used to improve dynamic IP addresses. In particular, where the addresses or locations are dynamic or not known to the client computer, the HTTP connections could still be made with the results being sent back via HTTP. This aspect may be crucial to a web services type of environment. The change in the IP address or the location of the database server will not in anyway effect the client as long as the application server can connect and the database server is able to accept the connection. The change in the configuration of the database will not effect the HTTP-JDBC driver setup. For example, no additional classes are needed and no additional configuration for the application is required. The client only has to pass the right set of changed parameters for connecting which will be used by the application server to connect to the database.

The driver system may be used by intelligent agents to obtain database access. In addition, jini services are built using Java components and, therefore, this driver technology could be used in a jini services environment. The driver system in accordance with the invention may also benefit a distributed computing system. In particular, the technology would support distributed computing generally, across the Internet and through firewalls and other security mechanisms.

The HTTP JDBC driver system in accordance with the invention may also be used to enhance peer-to-peer systems. In particular, the driver technology may be critical in supporting peer-to-peer communications where a peer needs to go through a firewall to get data from a data

storage device. In fact, in Project Jxta (http://www.Jxta.org), which is Sun Microsystems's peer-to-peer specification and platform, there are specific discussions about firewalls and network address translator's (NAT's) and the fact that you need one peer on the outside and one peer on the inside of the firewall, both being in the same peer group and able to transfer code and/or data. The driver system in accordance with the invention may also benefit N-tiered or multi-tiered architectures wherein, regardless of the number of "clients" and "servers" or the number of connections, the driver system can be used to transport data sets across firewalls as many times as may be required in a complex series of connections.

The HTTP JDBC driver system in accordance with the invention may be used with object-oriented databases and other new data storage devices. For example, the driver system will support any new data storage device, including existing types such as object-oriented databases (CloudScape), so long as the proper drivers exist which can be loaded onto the application server. This could become of critical importance in a web services environment, because the client would not be altered by the addition of new data storage devices or means. In addition, data could be pulled from such databases as PointBase.com (a Java relational database that has such a small footprint it can sit on a device, such as a router or vending machine).

The HTTP JDBC driver system in accordance with the invention may also be used with jini services or Web services. In an environment of the future built on jini services or web services, an application would be pulled together dynamically as components being assembled on the fly to create one-time applications. The driver system could act as a component to the services to provide dynamic access to external disparate data sources. Alternatively, the driver of this invention could be provided as its own service.

Thus, in accordance with the invention, a system for accessing data located behind a security mechanism is provided. The system comprises a client that executes a first application having a series of instructions that includes accessing data from a database wherein the client creates one or more database proxy objects that are used to indirectly access the database. The system further comprises an application server that executes a second application that interacts

with the database and has one or more corresponding objects, the application server further comprising one or more drivers that interface with a database so that the application server requests data from the database. The system causes the client to interact with the database through the application server so that a security mechanism protecting the client does not interrupt the accessing of the data from the database.

The system comprises a client that executes a Java applet having a series of instructions to access data from a database and creates proxy objects in response to the database request. The system further comprises a Java application server that executes a servlet that receives the calls generated by the proxy objects on the client and further generates database calls using the JDBC drivers.

The application server further comprises a mechanism to send back the results and the unique name of the object, if a new object is created on the server, or the resultset created by the database call. Such a system comprised of the above-mentioned components will make it possible for the applet to communicate with the database through the application server with the help of proxy objects, and by doing so the security mechanism protecting the client does not interrupt the database access and the JDBC functionality is achieved.

In accordance with yet another aspect of the invention, a system for accessing data located behind a security mechanism is provided wherein the system comprises a client having means for creating one or more proxy objects in response to a database request. The system further comprises an application server comprising means for processing the calls received from the database proxy objects and means for using one or more drivers to generate one or more database requests based on the calls from the proxy objects wherein the client interacts with the database through the application server so that a security mechanism protecting the client does not interrupt the accessing of the data in the database.

In accordance with yet another aspect of the invention a system for accessing data by a Java applet wherein the data is located behind a security mechanism is provided. The system comprises a client that executes a Java applet having a series of instructions to access data from a

database and creates proxy objects in response to the database request. The system further comprises a Java application server that executes a servlet that receives the calls generated by the proxy objects on the client and further generates database calls using the JDBC drivers. The application server further comprises a mechanism to send back the results along with a unique name of the object created by the database call, if any, or sends the resultset back to the client. Such a system comprised of the above-mentioned components will make it possible for the applet to communicate with the database through the application server so that a security mechanism protecting the client does not interrupt the database access.

Brief Description of the Drawings

Figure 1 is a block diagram illustrating a preferred embodiment of the secure data accessing system in accordance with the invention;

Figure 2 illustrates more details of the system shown in Figure 1;

Figure 3A illustrates a method for data accessing in accordance with the invention using the HTTP protocol and JDBC drivers;

Figure 3B is a block diagram illustrating the method for using the HTTP JDBC driver in accordance with the invention;

Figure 4 illustrates a typical JDBC client database connection;

Figure 5 illustrates a connection to a database using the HTTP JDBC driver in accordance with the invention; and

Figure 6 illustrates an application server being used as the client for another server in accordance with the invention.

Detailed Description of a Preferred Embodiment

The invention is particularly applicable to a distributed application system that uses Java applets, the HTTP protocol and JDBC drivers and it is in this context that the invention will be described. It will be appreciated, however, that the system and method in accordance with the invention has greater utility, such as to other computing systems wherein users behind security mechanisms need to access external data or where users need to access data located behind a security mechanism. The invention may also be implemented with other protocols or with other programming languages without departing from the scope of the invention.

Figure 1 is a block diagram illustrating a preferred embodiment of the secure data accessing system 20 in accordance with the invention. In more detail, the system may include one or more clients 22 (Client 1, Client 2, ..., Client N), and a distributed computer application system 24. The various elements of the system, such as the clients 22 and the distributed computer application system 24 may be interconnected together by a communications network 26 that permits all of the elements of the system to communicate with each other using a communications protocol. In a preferred embodiment, the communications network may be a computer network and more preferably the Internet and the World Wide Web. In a preferred embodiment, the communications over the communications network may be carried out using a well known protocol, such as the hypertext transfer protocol (HTTP). It may also be carried out using a secure communications protocol such as the well known secure hypertext transfer protocol (HTTPS).

Each client 22 may permit an authorized user of the system (e.g., someone with the appropriate access privileges) to access the resources of the distributed computer application system 24 as will be described in more detail below. Each client 22 may comprise a typical computer system that may include a display device 32, such as a CRT or LCD, a computer chassis 34 and one or more input/output devices 36, such as a keyboard 38 and a mouse 40 as shown. Each client may also include an output device, such as a printer (not shown) for generating a hard copy of the results of the distributed computer application being executed in

accordance with the invention. The reports may also be displayed on the display device 32. The computer chassis 34 may further include a central processing unit (CPU) 42 that controls the operation of the computer system, a temporary memory 44, such as DRAM or SRAM, that stores one or more pieces of software, such as the operating system, that are being currently executed by the CPU and a persistent storage device 46, such as a hard disk drive, optical disk drive, tape drive or the like, that permanently stores data and software. In order to access the distributed computer application system in accordance with the invention, a browser application 48 may be loaded from the persistent storage device into the memory as shown and executed by the CPU. The browser application, as is well known, permits the computer to connect to and communicate over the Internet 26 with other computer systems, such as the central computer application system 24. For example, the browser may communicate using a typical protocol, such as the well known HTTP protocol or the HTTPS secure protocol as described above. In general, the browser initiates a communications connection with the computer system and then establishes the protocol. In the preferred embodiment, the browser 48 permits a user of the client system 22 to access the system, execute one or more distributed applications using the distributed computer application system and receive output from the distributed applications. Typically, the browser may display a series of user interface screens, such as web pages, that permit the user to interact with the distributed computer application system. In a preferred embodiment, the client must have, at a minimum, the Java runtime environment which typically includes the ability to download applets, to create resultset objects, and to communicate over the http protocol, among other things.

The distributed computer application system 24 may be one or more typical server computer systems in the preferred embodiment wherein the one or more servers are interconnected together and communicate with each other over the communications network 26. In this manner, the combination of servers required to generate a distributed software application may be dispersed from each other and, when necessary, connect to each other and transfer data between each other using the communications network. In the example shown, the distributed computer application system may include an application server 60 and a database server 62. In

operation, a Java applet may be downloaded from the application server to the client so that it can be executed by the client. The applet may generate requests that are communicated to the database server over the Internet to connect to and retrieve the data from different databases. In a preferred embodiment, the applet to be downloaded to the client should contain the HTTP-JDBC client classes (See Appendix A and the description below) and the common classes shared by both the server and client (See Appendix B and the description below). To accomplish this, the system may include a routing database that maintains the properties (e.g., URLs) of the various resources needed to execute the distributed computer application.

In a preferred embodiment, the application server 60 will be a Java application server which will have, at a minimum, an http server, a servlet engine, and the ability to host applets, among other things. The server should have the JDBC drivers for the database that the client tries to connect to and the servlet that the client communicates with. The server should also have the common set of classes required by our HTTP-JDBC driver (See Appendix B and the description below) as well as the server side classes (See Appendix C and the description below).

As shown, the database server 62 and CLIENT #N may be located behind a corporate security mechanism 64, such as a firewall that controls the access of users to the resources located behind the security mechanism and controls the access that users behind the security mechanism have to the Internet. Thus, as shown, the database server and the application server are separated from each other by the security mechanism that makes the execution of the distributed computer application more difficult. In a preferred embodiment, the database or datasource should be able to accept connection from the Java application server using the appropriate JDBC drivers, with little or no interference from security measures or firewalls. In addition, the client (CLIENT #N) and the distributed computer application system 24 are also separated from each other by the security mechanism that also makes the execution of the distributed computer application more difficult as will now be described.

In particular, the clients 22 that need a database connection to the database server 62 to execute a distributed computer application may not be able to connect to the server from behind

secure corporate intranets. In more detail, since corporate intranets in general are secured with firewalls, proxies and other security measures, the clients will have to tunnel through them to gain access to the external resources. The problem is that most firewalls do not allow direct Internet Protocol (IP) traffic between the Internet and the internal network they are protecting. For example, most organizations behind firewalls have a well known proxy server running that allows people inside the organization to access the Web and, database servers.

The distributed computer application system typically needs extensive access to a variety of database servers that contain the necessary data to execute the distributed computer application. In general, JDBC drivers use a set of unique parameters for connecting to the database server and database servers, like all other servers, require the clients to connect to them through a particular port using client libraries and/or JDBC drivers. All JDBC drivers are required to conform to the JDBC API specifications developed by Sun Microsystems, Inc. Since all JDBC drivers are written to Java specifications all of them must implement the JDBC interfaces. The distributed computer application system in accordance with the invention connects and accesses data from databases using the JDBC drivers developed by the database vendors or third party developers. While connecting to the database server from the client seems to be an easy solution, JDBC drivers provided by the database vendors connect to the database using sockets and Java applets are restricted from making direct socket connections to other servers.

In a distributed computer application environment, the Java applets that may be used by the distributed computer application system to execute the application, need to communicate with the database server 62. However, Java applets are only allowed to communicate with the server that the applet is originated or downloaded from due to security restrictions. In addition, in a distributed computer application environment, it is more likely that the database server might be on a different server than the application server as shown in Figure 1. These servers may only be reachable by the client using the HTTP protocol. Thus, connecting to the different servers may be difficult and not all the servers are reachable by the clients because they may be behind firewalls and proxy servers.

Thus, to establish a connection between the client and the one or more servers of the distributed computer application system, the HTTP protocol is the common protocol used to communicate with the servers and the best way for the applets to communicate with the server is by using HTTP protocol to be as compatible as possible with the proxy servers and network firewalls. The details of this type of secure connection will now be described in more detail.

Figure 2 illustrates more details of the system shown in Figure 1 wherein the client 22 is executing a database distributed computer application and needs to access data stored on a database server 62 wherein the client is behind a corporate firewall 64. In more detail, each client 22 may further include a database client application 70, that may be one or more Java applets downloaded from the application server 60 and being executed by the client computer and a database proxy object 72 that communicates with the application server 60 as shown to request data. The application server 60 may further include a set of Java servlets 74 (which are Java applets that are executed by the application server and whose user interface is served to the client computer as one or more web pages as is well known) and a set of database drivers 76 that access the data stored in the database server 62. In the example herein, the database drivers may be JDBC drivers. The set of servlets 74 help create the connections between the client and the application server and the database server, execute the database statements/queries on the application server, and return the database query results back to the client. The client interacts with the application server using the standard HTTP protocol.

In a preferred embodiment, other Java classes required both on the server and the client are the javax.sql package provided by JavaSoft. The common classes required by both the client and the server should be the same version of compiled classes as a requirement for serializing and deserializing. Now, an example of the method for accessing data in accordance with the invention will be described.

Figure 3A illustrates an example of a method 100 for accessing data in accordance with the invention. In particular, an applet being executed by the client as part of the database client application 70 may make a request to the servlet 74 using the HTTP protocol with an object

name, a method name, and the parameters required to execute the method in step 102. The Servlet 74 in turn responds to the request by executing the method in step 104 (including accessing the database server). The servlet may then receive the results of the query in step 106 and pass the results back to the client in step 108 using the HTTP protocol. In accordance with the invention, HTTP-JDBC clients (as described below) together with Java's JDBC 2.0 API extension classes provide a better way for the applet to communicate with the servlets and provide the clients with the database driver functionality. Now, each portion of the system shown in Figure 2 will be described in more detail.

To execute the distributed computer application in accordance with the invention, the application server 60 in accordance with the invention must have all of the drivers for the databases since these will not be downloaded to the clients for use by the applets. Thus, each client will not make connections directly to any database server so that the security problems may be avoided. In turn, the applets on the client may communicate with the servlets to do all of the database calls. The set of appropriate proxy objects 72 will communicate with the actual objects on the server as necessary.

On the server side, these driver classes 76 require the set of servlets 74 for creating and managing the objects created by calls from the HTTP-JDBC clients. These servlets use Java's reflection to reflect the specified public method of the object, and execute it, and return the results to the client. If a resultset is to be returned to the client, then a javax.sql.CachedRowSet object is created and returned to the client. The CachedRowSet object implements both a resultset and serializable interfaces which makes it easy to serialize it for transporting it back to the client and the client can treat it as a resultset.

All the objects created on the server are added to a hashtable with their unique name as the key. When the client makes a request to execute a method on an object by passing the unique name, the servlet gets it from the hashtable. These objects are available on the server as long as the session is active. Once the session is invalidated, the objects created in the session are closed, if they need to be closed, and they are removed from the session and the database

connection is disposed off thus freeing all the resources associated with the database connections. These closed objects are then removed from the hashtable and will be available for garbage collection.

Each client 22 uses the typical HTTP protocol to communicate with the application server. The classes provided by the database vendor may not be serializable and may not be passed to the server and vice-versa. Thus, in this model, the client 22 may create the set of proxy objects 72 whose methods will mirror the methods of the JDBC objects created on the server. The client may then use these proxy objects 72 to communicate with the servlets 74. A new HTTP session is created on the client for each call to the DriverManager.connect() object. This object is the proxy for the actual session object on the server used later to execute methods and create statements on the server.

Since there is always a delay in connecting to the server via HTTP, the calls from the client to the server are batched for better performance. When a batch of statements are sent to the server for execution, the server executes them in the sequence they are created and returns the resultset along with the unique name of the actual object on the server. The proxy object 72 holds this name for contacting the object at a later time for executing another batch of methods on the same object. Depending on the method executed, the server will return either a resultset such as a unique name of a new object (e.g., Connection, Statement, PreparedStatement etc.) created on the server, or a null object. All the exceptions thrown by the server are serialized and sent to the client, if they should occur. If the execution of the method results in creating a new object on the server, then the unique name of the object is returned to the client. By overriding the finalize() method, as the proxy objects are garbage collected on the client, the actual objects will also be removed from the server and are also garbage collected.

Since the actual objects are created and held on the server, as required by the client, the proxy objects on the client are responsible for releasing them when they are not required. The calling of the close() method on the JDBC statement object releases all the resources held by the object. The servlet on the server maintains a list of active objects and it becomes the job of the

servlet to remove all the references of the closed object to release all the resources on the server. When a close() method is called on an object, the servlet closes the object and also removes it from the list. If an object has a close() method, then the proxy object's finalize() method is overridden to call close(), if it is not already closed. The finalize() method on Java object is called when the object is out of scope of the executing program, and has no references to it. This approach allows for better clean up of inactive objects on the server, thereby minimizing the memory requirements and maximizing the server performance. The functioning of the HTTP JDBC driver in accordance with the invention will be described in more detail now with reference to Figure 3B.

Figure 3B is a block diagram illustrating a preferred embodiment of the method for using the HTTP JDBC driver in accordance with the invention wherein the client 22, the application server 60 and the database server 62 are shown along with the communications network 26. The diagram illustrates the interaction of the various Java classes that implement the method. To understand this diagram, a brief description of the Java classes is now provided. A more detailed description of the Java classes is provided below.

Client Classes

These classes are downloaded as part of the applet to the client. These classes are used to create proxy objects for communicating with the server. These classes are responsible for serializing the object that encapsulates the database calls and de-serialize the results received from the server.

Common Classes

Since the classes that implement the JDBC interfaces are not serializable, the client needs to communicate with the server using serialized objects. For serialization to work, some of the same classes are needed, for serializing and deserializing, on both the client and server. These set of common classes are to be installed on the server, and the same classes are to be

downloaded to the client along with the applet. These set of classes also consist of JDBC 2.0 extension classes.

Server Classes

These classes are available on the application server and delegates the database calls between the database server and the applet. These classes primarily consist of a servlet and the supporting classes for processing the HTTP-JDBC calls from the client. These classes are responsible for de-serializing the object that encapsulates the database calls, executes the database call using the JDBC driver installed on the server, and returning back the results by serializing them.

Operation of Objects

While client and server objects function in isolation, the objects created using common classes are transported between the applet and the servlet. The serialized objects are tunneled through using HTTP between the client and the Java application server. When a query is executed which causes to return a table of data, java.sql.ResultSet object is created and returned. The server side classes of the HTTP-JDBC driver also makes use of the JDBC extension classes like javax.sql.CachedRowSet, which is sent back to the client from the server. Java.sql.ResultSet interface do not extend java.io.Serializable interface and so may not be capable of serializing. So, to solve the problem with serializing result sets to the client we create a CachedRowSet object from the result set and serialize it and return to the client.

The reason for creating proxy objects is because the objects created on the server when a database call is made are not transportable to the client as a Java object. For an object to be transportable it should be able to be serialized and de-serialized. The server application creates a javax.sql.CachedRowSet from the java.sql.ResultSet. javax.sql.CachedRowSet creates a serializable object from the java.sql.ResultSet object and since this also implements java.sql.ResultSet interface, the client application can just treat it as an object of type java.sql.ResultSet. There are many factors to consider in determining whether or not to transport

objects and, in some instances, transporting objects is simply not viable. In these and other instances, it may be better to use proxy objects. Therefore, in the preferred embodiment of the invention, certain objects are transported and certain objects are proxied, as discussed herein.

Returning to Figure 3B, the different steps performed during the secure data access in accordance with the invention using the HTTP JDBC drivers are shown as numbers inside of boxes (e.g., 1 - 6) wherein there may be interactions between more than one object or class during each step. For example, in the first step, the HTTPJDBCConnection, the HTTPJDBCPreparedStatement, the HTTPJDBCStatement, the HTTPJDBCCallableStatement and the HTTPJDBCDriver communicate with the HTTPJDBCSession class as shown.

The following steps are performed wherein the interaction of the objects and classes are shown in Figure 3B.

1. An object of type InObject is created if it is a single call. A MethodList object is created if the call can be batched wherein the MethodList object consists of one or more InObjects for executing in a particular sequence on the server.

2. When a statement is executed on the client, the InObject is serialized and sent to the server for execution by the HTTPJDBCSession object which sets up the object for execution on the server.

3. The HTTPJDBCSession object calls the servlet (JDBCServlet) and passes the InObject to it. The Servlet receives the InObject along with the name of the actual object to execute the method on and the session in which the actual object exists. The servlet then retrieves the actual object from the list using the name and executes the method on the object.

4. The Servlet receives the return value from the actual object after executing the method.

5. The Servlet returns the value back to HTTPJDBCSession over the communications network using the HTTP protocol.

6. The HTTPJDBCSession object returns the value back to the proxy object which generated the method. The proxy object returns the results back to the calling program.

In more detail, the first step in a JDBC application is to create a database connection. This is done by loading the class of the appropriate driver. By doing so, all the connections are made by specifying a URL that will use the appropriate driver to make a connection. When an application wants to use the HTTP-JDBC driver, the HTTPJDBCDriver class is loaded. While making the connection the URL to be specified may have the protocol as httpjdbc or httpsjdbc. When a connection is created using this HTTP-JDBC driver the HTTPSession sends the URL and properties to connect to the database to the server. The connection is made on the server and the Connection object is given a unique name and is added to the list of objects on the server. This unique name may be sent back and the proxy object of type HTTPJDBCConnection is created on the client and the unique name may be the name of the newly created proxy object. Then the application on the client may use the connection to execute the sql statements.

HTTPSession object on the client, created when a driver is loaded, contacts the JDBCServlet on the application server and sends the InObject or MethodList object for processing. Application may start using the HTTPJDBCConnection object just like a JDBC connection object. It may start executing various methods on the Connection object, just like in a typical JDBC application. When a method called by the application is to return a result then the proxy object will send the method(s) to the HTTPJDBCSession. HTTPJDBCSession object then makes a connection to the servlet on the server and sends the method(s) in one call. The JDBCServlet executes all the methods on the object in the same sequence as they are called on the client and returns the results to the HTTPJDBCSession object which in turn may return this back to the proxy object as a return value. Since the application that made the call on the proxy object is expecting a return value, the proxy object returns it to the application.

The JDBCServlet will use the unique name of the object to get the object from the list and execute the method, or sequence of methods on the object. Since the object on the server are JDBC objects, these objects are responsible for performing the required actions by connecting to the database and returning the values back. The JDBCServlet send back the OutObject which has the result after executing the method(s) and also the unique name if the servlet results in creating a new JDBC object on the server.

The table below illustrates an example of the key proxy objects and other key client-side objects of the HTTPJDBC driver, in the left column, and the corresponding server-side objects of the typical JDBC driver, in the right column. The purpose of the following table is to show a relative comparison between typical key JDBC driver objects and the key objects of the present invention in order to demonstrate a comparison of function and the relative differences.

TABLE 1

Client Proxy Classes	Conventional JDBC Classes
HTTPJDBCDriver	java.sql.Driver
HTTPJDBCConnection	java.sql.Connection
HTTPJDBCStatement	java.sql.Statement
HTTPJDBCPreparedStatement	java.sql.PreparedStatement
HTTPJDBCCallableStatement	java.sql.CallableStatement
HTTPJDBCDatabaseMetadata	java.sql.DatabaseMetadata
CachedRowSet	java.sql.resultset

An example using the HTTP-JDBC Driver is shown below as a piece of code.

```
// creates a Connection object on the server
HTTPJDBCConnection connection = HTTPJDBCDriver.connect(URL, properties);
-- start batch
HTTPJDBCPreparedStatement pstmt = connection.prepareStatement(String stmt);
pstmt.setString(1, String);
pstmt.setInt(2, int);
pstmt.setDate(3, Date);
ResultSet rset = (ResultSet)pstmt.executeQuery(); // CachedRowSet object is returned
pstmt.close();
rset.close()
-- end batch.
```

Now, a technique for downloading the drivers in accordance with the invention will be described. The distributed computer application system requires a set of classes that are to be downloaded as part of the applet to the clients. The servlets must be registered on the server with a Servlet container. The registering of a servlet will deploy the servlet each time a server is started or restarted. A set of initial arguments can be provided to the servlet when it is registered.

Some servlet containers allows the registered servlets to work in the security framework of the server, by defining access controls. In addition, all of the drivers must be installed on the server and made available to the servlets. The drivers may be loaded by the servlet when required.

The details of the various Java classes used in the preferred embodiment will now be described. In the attached Appendices (Appendix A, Appendix B and Appendix C), the source code for the various novel classes being described herein is provided. The server side classes may include various classes. For example, the server side classes may include a JDBCServerlet wherein this servlet, that is registered on the server, is called from the client with the method or batch of methods to be executed on the application server and return the results back to the client. The server may also include a ServerSessions class which is an object on the server created for each client session that is responsible for holding the JDBCServerSession objects (described below) created by the servlet. The server may also include a ServerSessions class which holds the active JDBCServerSession object that is created for each HTTPJDBCSession on the client. The JDBCServerSession object maintains all of the objects created for a session in a list. This class has the methods to close an object, if it can be closed, and remove it from the list. A SQLServer class is a particular implementation showing how to connect to a SQLServer database, depending on the properties provided by the client using the appropriate driver. The server may also include an Oracle class which is a particular implementation showing how to connect to a Oracle database, depending on the properties provided by the client using the appropriate driver. The server may also include a DBConnectionFactory which is used by the JDBCServlet to create connections to different databases and return them as type java.sql.Connection. The server may also include a DBVendor class that provides a common interface for implementing how to connect to a database using the driver provided by a database driver vendor.

In a preferred embodiment, there may be a number of shared classes that are shared between the client and the server. These classes may include an InObject which is a class that is responsible for creating the types for each parameter and is sent to the server along with the values for the parameters. The servlet takes this object and finds the correct object from the list

depending on the session name and object name. The server then executes the method this object represents and returns the value back to the client. An object of the type InObject is created for each method call. The classes may further include a MethodList class that extends java.util.ArrayList and holds all the methods to be executed as a batch. Each element in this list is of type InObject. When the server completes executing the batch it adds a name to this MethodList and sends back to the client along with the unique name. A NullObject is also included in the common objects since a null value cannot be serialized and transported back to the client. In particular, this object will be returned when a method call results in returning a null object. The common classes may further include an OutObject which is returned to the client along with a unique name and the resultset.

In a preferred embodiment, there may be a number of client classes that reside on the client. These classes may include a HTTPJDBCCallableStatement class that extends java.sql.PreparedStatement and implements the java.sql.CallableStatement interface. This proxy object is created on the client when the HTTPJDBCConnection.prepareCall(...) is executed. The actual object of type CallableStatement is created on the server and the unique name is sent back to the proxy object on the client which holds it. The client may also include a HTTPJDBCConnection class that extends JDBCStub (see below) and implements java.sql.Connection interface. This proxy object is created on the client when the HTTPJDBCDriver.connect(.....) is executed. The actual object of type java.sql.Connection is created on the server and the unique name is sent back to the proxy object on the client which holds it. The client may further include a HTTPJDBCDatabaseMetaData class that extends JDBCStub and implements java.sql.DatabaseMetaData interface. This proxy object is created on the client when the HTTPJDBCConnection.getMetaData() is executed. The actual object of type DatabaseMetaData is created on the server and the unique name is sent back to the proxy object on the client which holds it. The client further includes a HTTPJDBCDriver class that implements the java.sql.Driver interface. All of the connections created using this driver will be of type HTTPJDBCConnection. This follows the standard procedure for writing JDBC driver

java.sql.Driver specifications from Java Soft. This creates a HTTPJDBCSession on the client and also opens a communication channel with the server for executing all JDBC calls.

The client may further include a HTTPJDBCPreparedStatement class that extends HTTPJDBCStatement and implements java.sql.PreparedStatement interface. This proxy object is created when the HTTPJDBCConnection.prepareStatement(....) is executed. The actual object of type PreparedStatement is created on the server and the unique name is sent back to the proxy object on the client which holds it. A HTTPJDBCSession is the object that is responsible for serializing and deserializing the InObject or the MethodList depending on the type of call that is made and sent to the server and the OutObject or CachedRowSet object that is sent back. A HTTPJDBCStatement is a class that extends JDBCStub and implements java.sql.Statement interface. This proxy object is created when the HTTPJDBCConnection.createStatement() is executed. The actual object of type Statement is created on the server and the unique name is sent back to the proxy object on the client which holds it. A JDBCStub is a client side class only that has a set of methods that is extended by any class that needs to hold a unique name. Now, to better understand the invention, the differences between the typical JDBC driver and the HTTP JDBC driver in accordance with the invention will now be described.

The differences between the typical JDBC driver and the HTTP JDBC driver in accordance with the invention will illustrate the advantages and benefits of the HTTP JDBC driver in accordance with the invention. For example, the applet requirements for connection to a database is different. In particular, for the typical JDBC drivers, the drivers have to be downloaded to the client along with the applet. This is problematic since the drivers may be bulky and the client may take a long time to download all of the required classes to start using the driver. If the client needs to connect to different databases with different drivers, then all of the compatible drivers have to be downloaded at invocation of the applet. If the database does not have a pure JDBC driver, then additional software may be required to be installed on the client with the applet is invoked. Using the HTTP JDBC drivers in accordance with the invention, only a small set of classes needs to be downloaded along with the applet to the client. In addition, since connecting to multiple databases is made possible on the server side, none of

those drivers to connect to the multiple databases needs to be downloaded to the client. Thus, the HTTP JDBC driver alone will make it possible to connect to different database servers. This model in accordance with the invention will even support JDBC drivers which require additional software, since they will be installed on the server itself instead of the client. The HTTP JDBC driver in accordance with the invention provides the faster downloading of applets to the clients since these drivers make the clients thin, when it comes to communicating with the database servers.

The typical JDBC drivers also differ from the HTTP JDBC drivers in their protocol requirements. In particular, the JDBC drivers typically use proprietary protocols for communicating with the server so that each vendor may have different network protocol requirements. Thus, each different database may require its own protocol and hence must be downloaded to the client applet. In contrast, the HTTP JDBC driver uses the HTTP protocol to proxy the calls over HTTP to the application server layer. Then, the JDBC drivers available on the server are responsible for communicating with the database server and returning the results. The HTTP JDBC driver in accordance with the invention thus requires no special configuration at the client level. In addition, no matter what protocol the driver actually uses, that specific protocol is handled by the application server and the protocol is only utilized between the application server and the database server.

The typical JDBC driver also differs in the way that it may establish a connection as opposed to the HTTP JDBC drivers. In particular, the typical JDBC drivers connect directly to the server from the downloaded applet, if all the security, firewall and proxy requirements are met. The JDBC connection is based on parameters supplied to the database driver. The problem is that each database vendor uses different protocols and ports to connect to their database servers. In addition, due to the security restrictions in using applets, JDBC drivers downloaded to the client along with the applet may not be able to communicate with the database server because the database server may not be on the same host as the applet that was downloaded to the client so that the applet is not permitted to communicate with the database server. Furthermore, protocol and port requirements to connect to the database servers may not be

satisfied in corporate environments. The HTTP JDBC driver in accordance with the invention connects to the database via the application server layer and uses the same protocol and port that the client uses to download the applet thus making it more secure. In addition, since the application server actually establishes the connection, this process may take less time due to the network speed and processing capabilities of the servers. The advantages of better security and faster response times result from the use of the standard well known HTTP protocol and its standard ports, such as port 80.

The typical JDBC drivers and the HTTP JDBC driver in accordance with the invention also differ in the technique for sending SQL statements for execution. In particular, for the JDBC driver, when a statement is created, it is sent to the server immediately for compiling, after which this is ready for execution. In contrast, for the HTTP JDBC driver, a statement is sent to the application server, when it needs to be executed, for compilation and execution in batch mode all at the same time, using the built in batching techniques wherein one or more database requests are combined together in a single request. The batching of the SQL statements reduces the number of calls to the application server, thereby improving the performance and speed of the system.

The typical JDBC driver and the HTTP JDBC driver in accordance with the invention also differ in the way that they process the results. In particular, for the typical JDBC driver, the client, after executing the SQL statement, receives the results from the database and processes them. In contrast, with the HTTP JDBC driver in accordance with the invention, the server executes the statement, receives and processes the results, creates a "serializable" object(CachedRowSet) and sends it using the HTTP protocol to the client. Thus, the processing of results and the computation load for the processing is shared by both server and client so that it may be executed more quickly. An example of the above differences will now be described with reference to Figures 4 and 5 to better illustrate those differences.

Figure 4 illustrates a typical JDBC client database connection system 120. In particular, an applet client 122 is shown that is located behind a firewall 124. With this typical

arrangement, the client applet may include the JDBC drivers which has the disadvantages set forth above. The system may also include a Java application server 126 and a database server 128 that interact with the client in order to retrieve data from the database. As shown, to accomplish this task using this typical system and connection, the applet and the Java application server may interact and communicate with each other over Port 80 as is well known. Then, the client applet must communicate directly with the database server using the JDBC drivers and a typical socket connection using port 1521 which is the usual port for database access. As described above, this configuration is problematic since the client applet may be unable to connect to the database server using port 1521 due to the firewall. In the alternative, it is necessary that the firewall is programmed to allow the applet to connect to the database server running on port 1521 which leaves a hole in the security of the firewall that can be exploited.

The following piece of code illustrates how a typical applet can connect to the database server using JDBC drivers that are downloaded along with the applet as shown in Figure 4.

```
// load the appropriate driver for connecting to the desired database server, Oracle
// Sybase, or SQL Server
Class.forName(" jdbc.DriverClassName");

// connect to the database at the specified URL with the login and password
// URL string specifies the database host, server name, port other required
parameters to connect to //the database.
Connection con = DriverManager.getConnection(URL, "myLogin",
"myPassword");
Statement stmt = con.createStatement();
Connection con = DriverManager.getConnection(URL, "myLogin",
"myPassword");
Statement stmt = con.createStatement();
ResultSet rset = stmt.executeQuery("select * from table");
While (rset.next()){
   String col1Value = rset.getString("column1");
   // get the values from the resultset
}
```

Now, the HTTP JDBC driver in accordance with the invention that has the above described advantages and benefits over the typical client applet with JDBC drivers will be described.

Figure 5 illustrates a connection to a database using the HTTP JDBC driver system 130 in accordance with the invention. In particular, an applet client with the HTTP JDBC drivers 132 is shown behind a firewall 134. In particular, the firewall separates the applet client from an Java application server 136. To request data from a database server 138, the applet client may generate an HTTP request that passes through the firewall to the application server over the typical port 80. The application server, with JDBC drivers installed in accordance with the invention, may then generate a database request and then create a socket connection over port 1521 to the database server as shown without passing through the firewall. The application server may receive the response from the database server and then forward the response back to the applet client using the typical HTTP protocol. As shown, with the system in accordance with the invention, the applet client does not have to attempt to access the database server directly and the socket connection between the application server and the database server is not interfered with by the firewall.

The following piece of code illustrates how a HTTP JDBC applet in accordance with the invention requests data from the database server using the HTTP JDBC drivers in accordance with the invention.

```
// load the HTTP-JDBC Driver using the appropriate class name
Class.forName("com.VSDV.HTTPJDBC.Client.Driver");

// connect to the database at the specified URL with the login and password
// URL string specifies the protocol, database host, server name, port other
required
// parameters to connect to the database
// While using the HTTP-JDBC driver the protocol should be httpjdbc or
httpsjdbc
Connection con = DriverManager.getConnection(URL, "myLogin",
"myPassword");
Statement stmt = con.createStatement();
ResultSet rset = stmt.executeQuery("select * from table");
While (rset.next()){
    String col1Value = rset.getString("column1");
    // get the values from the resultset
}
```

The HTTP JDBC driver system in accordance with the invention may be used with various different JDBC drivers. In particular, since the driver system in accordance with the invention uses proxy objects to communicate with the application server, any type of JDBC driver can be supported without any change to the existing code. In other words, all of the JDBC driver types can be supported without having the clients download any code so that no installation of the drivers is required on the client. Thus, the HTTP JDBC driver system is adaptable and can be used with any past, current or future JDBC driver. In addition, the servlets of the application server that are executed when a JDBC call is made can use any type of driver available for a specific platform. This driver can support any or all the types of drivers that are available for the platform that the application server is installed. These types of drivers may include the following:

Type 1 - JDBC-ODBC bridge plus ODBC driver;

Type 2 - Native-API partly-Java driver;

Type 3 - JDBC-Net pure Java driver; and

Type 4 - Native-protocol pure Java driver.

For more information on these types of drivers refer to the driver types documentation page at http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/intro.html#1018502. Now, the advantages of using the HTTP JDBC driver system in different scenarios will be described in more detail.

The downloading of the code to the client as part of the applet and creating proxy objects on the client for communicating with the JDBC objects gives the clients enormous power during runtime. For example, a client may dynamically connect to any database through the application server if the suitable drivers are loaded on the server. Since the JDBC communication is proxied by the client, all that is needed are the set of parameters for connecting to a database server. If a JDBC compliant driver can be found for the Java application server then the client can create a proxy object to communicate with the database connection object on the server. In fact, regardless of the protocol and communication requirements for a particular database, the client code does not change and no additional code will need to be downloaded for connecting to a different database. In addition, the client side of the driver can delegate calls to any database provided the application server has the suitable driver for connecting to the database. Unlike traditional JDBC drivers that are different for each different database, the HTTP JDBC driver in accordance with the invention remains the same for accessing any database. A first scenario that may use the HTTP JDBC driver in accordance with the invention is a client/server system.

In this scenario, the database server and the Java application server are both on the same physical machine/computing resource. The client that is accessing the application is behind a firewall. The firewall allows only HTTP protocol communications and permits connections to a particular port, typically port 80. Both the web service provided by the Java application server and the database server cannot listen for incoming requests on the same port and both the services use different protocols. Thus, even though the applet is allowed to make socket connections to the server it is downloaded from, the firewall might prevent this from occurring when using a traditional JDBC driver. Thus, the HTTP-JDBC driver can be very useful in this scenario.

In another scenario, the database server and the application server are on separate physical machines and have different IP addresses. This is the scenario that was shown in Figure 2 and described above. In another scenario, the application server may be a client for another application server. Figure 6 illustrates a system 140 wherein an application server 142 is being used as the client for another server in accordance with the invention. In particular, a client 144 may connect and communicate with the application server 142 wherein the client and application server are both on the same side of a firewall 146. On the other side of the firewall is a second application server 148 and a database server 150. In more detail, the Java application server 142 behind a firewall would become a client to another Java application server 148 that is outside the firewall and has access to the database 150. This architecture is very useful for server side programs requiring database access to disparate data sources wherein all the security restrictions can be overcome by using the HTTP-JDBC driver in accordance with the invention. In one embodiment, the HTTP JDBC driver system may be used by all the J2EE components that are provided by the Java application server that requires database access.

In another scenario, the HTTP JDBC driver in accordance with the invention may be used effectively in distributed computing. In particular, the HTTP-JDBC driver will provide a solution for connecting to disparate and/or external data sources in a distributed computing environment. Since the HTTP JDBC driver is based on the most common Internet communication protocol, the driver can be used by any device in a network that has a Java runtime environment with HTTP protocol support. In addition, since the technology for distributed computing using Java components is emerging and evolving rapidly, any Java component in a Java enabled distributed environment can efficiently use HTTP-JDBC driver to gain access to disparate data sources

In another scenario, the HTTP JDBC driver may be used with Jini Services. The Jini technology enables open end-to-end solutions for creating dynamically networked products, services, and applications that scale from device to the enterprise. The HTTP-JDBC driver may be used to provide dynamic database access to a wide variety of databases and other data sources that are required by a Jini service. Since Jini clients can be behind secure networks, the same

problems, explained earlier, arise when trying to connect to the database using a protocol and a port that is prohibited by the firewall. Because of the advantages this has over typical JDBC drivers, explained earlier, this will provide a better solution both in terms of downloading a small piece of code and delegating all the database access to the Java application server that is outside the firewall and has access to the database server. This driver can be snapped in to a service dynamically and provide the database access without the communication restrictions of the firewall. In this scenario the database connectivity parameters may be provided dynamically depending on the client requiring access to the database.

In another scenario, the driver may be used with peer-to-peer computing systems such as project JXTA. In particular, in a Java peer-to-peer environment, the driver in accordance with the invention may be used to access a disparate data source without having to download a lot of classes while using the most common internet protocol HTTP. Since JXTA does not specify any language requirements, this driver can support any two peers in which one peer has a Java virtual machine and the other has a Java application server. Since the HTTP-JDBC driver can work on clients with a small memory footprint this can be used by virtually any device in a peer group that has the Java runtime environment. Since JXTA specification does not specify that the peers be developed in a particular language for interoperability, this driver can support any peer in a peer-to-peer environment with a Java virtual machine and HTTP support. The peers can use this driver to gain access to a database outside of the firewall by connecting to a Java application server and delegating all the database communication to the server side components of the driver installed on the application server.

In more detail, Project JXTA addresses the need for an open, generalized protocol that inter-operates with any peer on the network including PCs, servers and other connected devices. The goal of project JXTA is to develop basic building blocks and services that would enable innovative applications for peer groups. A peer can offer a service by itself or in co-operation with other peers. In a peer-to-peer service model, if a service should provide access to a database, then using this driver would make access to the database simple and efficient. If a peer was to gain access to a database outside of the firewall then using this driver would make it

easier to penetrate the firewall using the most widely used protocol HTTP and port 80. In this environment the parameters to connect to a database might vary from service to service or even at a peer level, this driver provides an efficient way of gaining access to the database because of all it's virtues explained above.

In a peer group one of the peers can provide this driver as a part of the peer group service to connect to any database. The peer delivering this service component can have all the required drivers to connect to different databases or it can discover these drivers available on the network and provide the database access to the required database.

Additional details of the HTTP tunneling, servlets and session tracking used to implement the HTTP JDBC driver in accordance with the invention may be found at the locations below. The information located at those links are incorporated herein by reference.

HTTP Tunneling

http://developer.java.sun.com/developer/technicalArticles/InnerWorkings/Burrowing/index.html

http://developer.java.sun.com/developer/technicalArticles/InnerWorkings/JDCPerformTips/

Servlets and Session tracking

http://java.sun.com/docs/books/tutorial/servlets/client-state/session-tracking.html

http://java.sun.com/j2ee/tutorial/doc/Servlets11.html

http://java.sun.com/docs/books/tutorial/servlets/TOC.html

While the foregoing has been with reference to a particular embodiment of the invention, it will be appreciated by those skilled in the art that changes in this embodiment may be made

without departing from the principles and spirit of the invention, the scope of which is defined by the appended claims.